

On PGZ decoding of alternant codes

R. Farré, N. Sayols, and S. Xambó-Descamps

Universitat Politècnica de Catalunya

rafel.farre@upc.edu, narcissb@gmail.com, sebastia.xambo@upc.edu

Abstract

In this note we first review the classical Petterson-Gorenstein-Zierler decoding algorithm for the class of alternant codes (which includes Reed-Solomon, Bose-Chaudhuri-Hocquenghem and classical Goppa codes), then we present an improvement of the method to find the number of errors and the error-locator polynomial, and finally we illustrate the procedure with several examples. In two appendices we sketch the main features of the system [3] we have used for the computations.

Keywords: Alternant codes, RS codes, BCH codes, classical Goppa codes
2010 MSC: 11T71, 94B05, 94B35, 94B15

Introduction

The Petterson-Gorenstein-Zierler decoding algorithm (PGZ for short) was first developed for Reed-Solomon codes (RS), and later applied to Bose-Chaudhuri-Hocquenghem codes (BCH). In [4], two flavours of it were presented for alternant codes, with due attention to the computational aspects. The main interest of working with the class of alternant codes is that it includes many interesting subclasses, like RS codes, BCH codes (the most relevant class of cyclic codes), and classical Goppa codes. The practical bonus of this realization is that all these families of codes can be constructed by specializing the general constructor of alternant codes and, most fundamentally, that any effective decoding algorithm for alternant codes is sufficient (and effective) for all those subclasses.

In this note we present a natural improvement, both conceptual and computational, of the PGZ algorithm. The key point is that the output of the Gauss-Jordan reduction of a (Hankel-like) matrix constructed from the syndrome vector gives directly and at the same time the number of errors and the error-locator polynomial.

The organization is as follows. In the first section we briefly review alternant codes. This includes details about how the classes of codes just

mentioned can be constructed with special calls to the main constructor. The second section is devoted to present the mathematical basis of the PGZ approach for the decoding of alternant codes. Our improvement of PGZ is explained in detail in the third section and in the fourth we provide several examples. Finally Appendix A contains listings of the key Python functions that we have elaborated to get clear implementations of the computations and in the Appendix B we sketch the main features of the Python package [3] used to script the examples.

Notations and conventions. If q is a prime power, the finite field of q elements (unique up to isomorphism) is denoted \mathbb{F}_q . It is a subfield of \mathbb{F}_{q^m} for all positive integers m . The field \mathbb{F}_{q^m} can be constructed as the quotient $\mathbb{F}_q[X]/(f)$, where $f \in \mathbb{F}_q[X]$ is any irreducible polynomial over \mathbb{F}_q of degree m .

Given elements $\alpha_1, \dots, \alpha_n$ in a ring, we write $V_r(\alpha_1, \dots, \alpha_n)$ to denote the *Vandermonde matrix* of r rows associated to $\alpha_1, \dots, \alpha_n$. In other words, its rows have the form $(\alpha_1^i, \dots, \alpha_n^i)$ for $0 \leq i \leq r-1$. The determinant of the matrix $V_n(\alpha_1, \dots, \alpha_n)$, which is called the *Vandermonde determinant* of $\alpha_1, \dots, \alpha_n$, is equal to $\prod_{1 \leq i < j \leq n} (\alpha_j - \alpha_i)$. In particular, it is non-zero when the α_k are distinct elements of a field.

Let K be a finite field. A linear code of length n defined over K is a vector subspace $C \subseteq K^n$. If C has dimension k , we say that C is an $[n, k]$ code. The quotient k/n is called the *rate* of C . The *Hamming distance* $\text{hd}(y, y')$ of $y, y' \in K^n$ is the number of indices $j \in \{1, \dots, n\}$ such that $y_j \neq y'_j$. The *minimum distance* of a linear code C , denoted d , is the minimum of the distances $\text{hd}(x, x')$ for $x, x' \in C$, $x \neq x'$. The number of non-zero entries of $y \in K$ is called the *weight* of y and is denoted $\text{wt}(y)$. It is easy to see that d is the minimum of the weights of non-zero elements of C . An $[n, k]$ code of minimum distance d is said to be an $[n, k, d]$ code, or an $[n, k, d]_K$ if we need to write the base field K explicitly.

1. Essentials on alternant codes

Let $K = \mathbb{F}_q$ and $\bar{K} = \mathbb{F}_{q^m}$. Let $\alpha_1, \dots, \alpha_n$ and h_1, \dots, h_n be elements of \bar{K} such that $h_i, \alpha_i \neq 0$ for all i and $\alpha_i \neq \alpha_j$ for all $i \neq j$. Consider the matrix

$$H = V_r(\alpha_1, \dots, \alpha_n) \text{diag}(h_1, \dots, h_n) \in M_n^r(\bar{K}), \quad (1)$$

that is,

$$H = \begin{pmatrix} h_1 & \dots & h_n \\ h_1 \alpha_1 & \dots & h_n \alpha_n \\ \vdots & & \vdots \\ h_1 \alpha_1^{r-1} & \dots & h_n \alpha_n^{r-1} \end{pmatrix} \quad (2)$$

We say that H is the *alternant control matrix* of order r associated with the vectors

$$\mathbf{h} = (h_1, \dots, h_n) \quad \text{and} \quad \boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_n).$$

To make explicit that the entries of \mathbf{h} and $\boldsymbol{\alpha}$ (and hence of H) lie in \bar{K} , we will say that H is defined over \bar{K} .

The K -code $A_K(\mathbf{h}, \boldsymbol{\alpha}, r)$ defined by the control matrix H is the subspace of K^n whose elements are the vectors x such that $xH^T = 0$. Such codes will be called *alternant codes*. If we define the H -*syndrome* of a vector $y \in \bar{K}^n$ as $s = yH^T \in \bar{K}^r$, then $A_K(\mathbf{h}, \boldsymbol{\alpha}, r)$ is just the subspace of K^n whose elements are the vectors with zero H -syndrome.

1.1 Proposition (Alternant bounds). *If $C = A_K(\mathbf{h}, \boldsymbol{\alpha}, r)$, then*

$$n - r \geq \dim C \geq n - rm$$

and

$$d \geq r + 1$$

(minimum distance alternant bound).

Proof. See, for example, [4], p. 183. □

For the proofs of the statements in the remainder of this section, we refer to [4], Section 4.1.

Reed-Solomon codes

Given a list or vector $\boldsymbol{\alpha}$ of distinct non-zero elements $\alpha_1, \dots, \alpha_n \in K$, the Reed-Solomon code

$$C = \text{RS}(\boldsymbol{\alpha}, k) \subseteq K^n$$

is the subspace of K^n generated by the rows of the Vandermonde matrix $V_k(\alpha_1, \dots, \alpha_n)$. It turns out that

$$\text{RS}(\boldsymbol{\alpha}, k) = A_K(\mathbf{h}, \boldsymbol{\alpha}, n - k),$$

where $\mathbf{h} = (h_1, \dots, h_n)$ is given by

$$h_i = 1 / \prod_{j \neq i} (\alpha_j - \alpha_i). \tag{3}$$

Note that in this case $\bar{K} = K$, hence $m = 1$, and that the alternant bounds are sharp. Indeed, we have $r = n - k$, hence $k = n - r$, while $n - k + 1 \geq d$ (by the Singleton bound) and $d \geq r + 1 = n - k + 1$ by the minimum distance alternant bound. In other words, C is MDS (maximum distance separable).

An RS code is called *primitive* if the $\alpha_1, \dots, \alpha_n$ are all non-zero elements of K . In that case, a natural way to proceed is to generate those elements as the powers $1, \alpha, \dots, \alpha^{q-2}$ of a primitive element α of K and so the code is, if its dimension is k , $\text{RS}([1, \alpha, \dots, \alpha^{n-1}], k)$, where $n = q - 1$.

Generalized Reed-Solomon codes. The vector \mathbf{h} in the definition of the code $\text{RS}([\alpha_1, \dots, \alpha_n], k)$ as an alternant code is obtained from α by the formula (3). If we allow that \mathbf{h} can be chosen possibly unrelated to α , but still with components in K , the resulting codes $A_K(\mathbf{h}, \alpha, n - k)$ are called *Generalized Reed-Solomon* (GRS) codes, and we will write $\text{GRS}(\mathbf{h}, \alpha, k)$ to denote them. An argument as above shows that such codes have type $[n, k, n - k + 1]$. Notice that the code $A_K(\mathbf{h}, \alpha, r)$ is the intersection of the GRS code $A_{\bar{K}}(\mathbf{h}, \alpha, r)$ with K^n .

BCH codes

These codes are denoted $\text{BCH}(\alpha, d, l)$, where $\alpha \in \bar{K}$ and $d > 0, l \geq 0$ are integers (called the *design minimum distance* and the *offset*, respectively). When $l = 1$, we simply write $\text{BCH}(\alpha, d)$ and say that it is a *strict* BCH code. The good news here is that

$$\text{BCH}(\alpha, d, l) = A_K(\mathbf{h}, \alpha, d - 1), \quad (4)$$

where $\mathbf{h} = (1, \alpha^l, \alpha^{2l}, \dots, \alpha^{(n-1)l})$, $\alpha = (1, \alpha, \alpha^2, \dots, \alpha^{(n-1)})$, $n = \text{period}(\alpha)$.

If α is a primitive element of K , and hence $n = q - 1$, we have the equality

$$\text{BCH}(\alpha, n - k + 1) = \text{RS}([1, \alpha, \dots, \alpha^{n-1}], k).$$

Classical Goppa codes

Let $g \in \bar{K}[T]$ be a polynomial of degree $r > 0$ and let $\alpha = \alpha_1, \dots, \alpha_n \in \bar{K}$ be distinct non-zero elements such that $g(\alpha_i) \neq 0$ for all i . Then the *classical Goppa code* over K associated with g and α , which will be denoted $\Gamma(g, \alpha)$, can be defined as $A_K(\mathbf{h}, \alpha, r)$, where \mathbf{h} is the vector $(1/g(\alpha_1), \dots, 1/g(\alpha_n))$. Thus the minimum distance of $\Gamma(g, \alpha)$ is $\geq r + 1$ and its dimension k satisfies $n - rm \leq k \leq n - r$. The minimum distance bound can be improved to $d \geq 2r + 1$ in the case that $K = \mathbb{F}_2$ and the roots of g are distinct.

2. The PGZ decoding approach

Let $C = A_K(\mathbf{h}, \alpha, r)$ be an alternant code. Let $t = \lfloor r/2 \rfloor$, that is, the highest integer such that $2t \leq r$. For reasons that will become apparent below, t is called the *error-correction capacity* of C .

Let $x \in C$ (using a transmission channel terminology, we say that it is the *sent vector*) and $e \in \bar{K}$ (*error vector*, or *error pattern*). Let $y = x + e$ (*received vector*). The goal of a decoder is to obtain x from y and H when $l = \text{wt}(e) \leq t$. Henceforth we will assume that $l > 0$.

If $e_j \neq 0$, we say that j is an *error position*. Let $\{m_1, \dots, m_l\}$ be the error positions and $\{e_{m_1}, \dots, e_{m_l}\}$ the corresponding *error values*. The *error*

locators η_1, \dots, η_l are defined by $\eta_k = \alpha_{m_k}$. Since $\alpha_1, \dots, \alpha_n$ are distinct, the knowledge of the η_k is equivalent to the knowledge of the error positions.

The monic polynomial $L(z)$ whose roots are the error locators is called the *error-locator polynomial*. Notice that

$$L(z) = \prod_{i=1}^l (z - \eta_i) = z^l + a_1 z^{l-1} + a_2 z^{l-2} + \dots + a_l, \quad (5)$$

where $a_j = (-1)^j \sigma_j = \sigma_j(\eta_1, \dots, \eta_l)$ is the j -th elementary symmetric polynomial in the η_i ($0 \leq j \leq l$).

The *syndrome* of y is the vector $s = yH^T$, say $s = (s_0, \dots, s_{r-1})$. Since $xH^T = 0$, we have $s = eH^T$. Inserting the definitions, we easily find that

$$s_j = \sum_{i=0}^{n-1} e_i h_i \alpha_i^j = \sum_{k=1}^l h_{m_k} e_{m_k} \alpha_{m_k}^j = \sum_{k=1}^l h_{m_k} e_{m_k} \eta_k^j \quad (6)$$

We will use the following notations:

$$A_l = \begin{pmatrix} s_0 & s_1 & \dots & s_{l-1} \\ s_1 & s_2 & \dots & s_l \\ \vdots & \vdots & \ddots & \vdots \\ s_{l-1} & s_l & \dots & s_{2l-2} \end{pmatrix} \quad (7)$$

and the vector

$$\mathbf{b}_l = (s_l, \dots, s_{2l-1}). \quad (8)$$

Next proposition establishes the key relation for computing the error-locator polynomial.

2.1 Proposition. *If $\mathbf{a}_l = (a_l, \dots, a_1)$ (see the fomula (5)), then*

$$\mathbf{a}_l A_l + \mathbf{b}_l = 0. \quad (9)$$

Proof. Substituting z by η_i in the identity

$$\prod_{i=1}^l (z - \eta_i) = z^l + a_1 z^{l-1} + \dots + a_l$$

we obtain the relations

$$\eta_i^l + a_1 \eta_i^{l-1} + \dots + a_l = 0,$$

where $i = 1, \dots, l$. Multiplying by $h_{m_i} e_{m_i} \eta_i^j$ and adding with respect to i , we obtain (using (6)) the relations

$$s_{j+l} + a_1 s_{j+l-1} + \dots + a_l s_j = 0,$$

where $j = 0, \dots, l-1$, and these relations are equivalent to the stated matrix relation. \square

2.2 Remark. In the equation (9), the matrix A_l turns out to be non-singular and hence it determines \mathbf{a}_l (and $L(z)$) uniquely, namely $\mathbf{a}_l = -\mathbf{b}_l A_l^{-1}$. In next section we are going to establish this fact as a corollary of our Theorem 3.1, whose main outcome is *a fast solution of that equation*.

The roots of L only tell us in what positions the errors occur. To find the actual value of the errors, we need the syndrome polynomial, $\sigma(z) = s_0 + s_1 z + \dots + s_{r-1} z^{r-1}$ and the polynomial $\tilde{L}(z) = 1 + a_1 z + \dots + a_l z^l$. Notice that the roots of $\tilde{L}(z)$ are $1/\eta_1, \dots, 1/\eta_l$.

2.3 Theorem (Forney's formula). *Let $E(z) = \tilde{L}(z)\sigma(z) \pmod{z^r}$. Then for any $m \in \{m_1, \dots, m_l\}$ we have*

$$e_m = -\frac{\alpha_m E(1/\alpha_m)}{h_m \tilde{L}'(1/\alpha_m)}, \quad (10)$$

where $\tilde{L}'(z)$ denotes the derivative of $\tilde{L}(z)$.

Proof. See, for example, [4], sections 4.2 and 4.4. \square

Because of this result, the polynomial $E(z)$ is called the *error-evaluator* polynomial.

3. Fast PGZ computations

The main object considered in this Section is the matrix (cf. [1])

$$S = \begin{pmatrix} s_0 & s_1 & \cdots & s_{l-1} & s_l & \cdots & s_t \\ s_1 & s_2 & \cdots & s_l & s_{l+1} & \cdots & s_{t+1} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ s_{l-1} & s_l & \cdots & s_{2l-2} & s_{2l-1} & \cdots & s_{t+l-1} \\ \hline \vdots & \vdots & & \vdots & \vdots & & \vdots \\ s_{t-1} & s_t & \cdots & s_{t+l-2} & s_{t+l-1} & \cdots & s_{2t-1} \end{pmatrix} \quad (11)$$

Note that $2t-1 \leq r-1$, so that all components are well defined. Note also that the $l \times l$ submatrix at the upper left corner is the matrix A_l defined

in (7) and that the column $(s_l, s_{l+1}, \dots, s_{2l-1})^T$ to its right is the vector \mathbf{b}_l defined in (8).

In next Theorem we use the following notation: $V_s = V_s(\eta_1, \dots, \eta_l)$. Thus the i -th row of V_s , for $0 \leq i \leq s-1$, is the vector $(\eta_1^i, \dots, \eta_l^i)$. We also write $D = \text{diag}(h_{m_1}e_{m_1}, \dots, h_{m_l}e_{m_l})$.

3.1 Theorem. $S = V_t D V_{t+1}^T$.

Proof. Let $0 \leq i \leq t-1$ and $0 \leq j \leq t$. Then the j -th column of $D V_{t+1}^T$ is the column vector $(h_{m_1}e_{m_1}\eta_1^j, \dots, h_{m_l}e_{m_l}\eta_l^j)^T$. It follows that the element in row i column j of $V_t D V_{t+1}^T$ is $h_{m_1}e_{m_1}\eta_1^{i+j} + \dots + h_{m_l}e_{m_l}\eta_l^{i+j} = s_{i+j}$ (by the equation (6)). \square

3.2 Corollary. *The rank of S is l and the matrix A_l is non-singular.*

Proof. Since D has rank l , the rank of S is at most l . On the other hand, the theorem shows that $A_l = V_l D V_l^T$ and therefore

$$\det(A_l) = \det(V_l)^2 \det(D) \neq 0.$$

Note that $\det(V_l)$ is the Vandermonde determinant of η_1, \dots, η_l , which is non-zero because the error locators are distinct. \square

3.3 Corollary. *The Gauss-Jordan algorithm applied to the matrix S returns a matrix that has the form*

$$\left(\begin{array}{cccccc} 1 & 0 & \cdots & 0 & -a_l & * \\ 0 & 1 & \cdots & 0 & -a_{l-1} & * \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -a_1 & * \\ \hline \vdots & \vdots & & \vdots & \vdots & \vdots \end{array} \right) \quad (12)$$

where $*$ denotes unneeded values (if any) and the vertical dots below the horizontal line denote that all its elements (if any) are zero. This matrix gives at the same time l , the number of errors, and the coefficients of the error-locator polynomial. \square

Putting together what we have learned in the last two sections, we obtain two algorithms to decode alternant codes, or rather two variants of an algorithm. We call them **PGZ** and **PGZm**, for in essence they are due to Peterson, Gorenstein and Zierler (see [2]). They share the same scheme for finding the location of the errors, but differ in how the error values are computed. PGZm is the simplest of the two, as it relies mainly on linear algebra,

whereas PGZ relies on the finding the error evaluator polynomial and using Forney's formula.

In the descriptions that follow, *Error* means “a suitable decoding-error message” and the function $\mathbf{GJ}(\mathbf{S})$ return the values $-a_l, \dots, -a_1$ of the matrix (12) as a column vector (this is a slightly modified form of the Gauss-Jordan procedure). In detail, it works as follows:

Improved PGZ

1. Get the syndrome vector, $s = (s_0, \dots, s_{r-1}) = yH^T$. If $s = 0$, return y .
2. Form the matrix S as in the equation (11).
3. Set $\mathbf{a} = -\mathbf{GJ}(S)$ (equation (12)). After this we have a_1, \dots, a_l , hence also the error-locator polynomial L .
4. Find the elements α_j that are roots of the polynomial L . If the number of these roots is $< l$, return *Error*. Otherwise let η_1, \dots, η_l be the error-locators corresponding to the roots and set $M = \{m_1, \dots, m_l\}$, where $\eta_i = \alpha_{m_i}$.
5. Let $\sigma(z) = s_0 + s_1z + \dots + s_{r-1}z^{r-1}$, $\tilde{L}(z) = 1 + a_1z + \dots + a_lz^l$ and compute the error-evaluator polynomial by the formula

$$E(z) = \tilde{L}(z)\sigma(z) \mod z^r.$$

6. Find the errors e_m , for all $m \in M$, using Forney's formula (equation (10)). If any of the values of e_m is not in K , return *Error*. Otherwise return $y - e$.

3.4 Theorem. *The algorithm PGZ corrects up to t errors*

Proof. It is an immediate consequence of what we have seen so far. \square

3.5 Remark. In step 5 of the algorithm we could use the alternative syndrome polynomial $\tilde{\sigma}(z) = s_0z^{r-1} + s_1z^{r-2} + \dots + s_{r-1}$, find the alternative error evaluator E^* as the remainder of the division of $L(z)\tilde{\sigma}(z)$ by z^r and then, in step 6, use the following alternative Forney formula ([4], P.4.9):

$$e_m = -E^*(\alpha_m)/h_m\alpha_m^r L'(\alpha_m). \quad (13)$$

Algorithm PGZm

The steps 5 and 6 of the PGZ algorithm can be compressed into a single step consisting in solving for e_{m_1}, \dots, e_{m_l} the following system of linear equations:

$$h_{m_1}e_{m_1}\eta_1^j + h_{m_2}e_{m_2}\eta_2^j + \dots + h_{m_l}e_{m_l}\eta_l^j = s_j \quad (0 \leq j \leq l-1),$$

which is equivalent to the matrix equation

$$\begin{pmatrix} h_{m_1} & h_{m_2} & \dots & h_{m_l} \\ h_{m_1}\eta_1 & h_{m_2}\eta_2 & \dots & h_{m_l}\eta_l \\ h_{m_1}\eta_1^2 & h_{m_2}\eta_2^2 & \dots & h_{m_l}\eta_l^2 \\ \vdots & \vdots & \ddots & \vdots \\ h_{m_1}\eta_1^{l-1} & h_{m_2}\eta_2^{l-1} & \dots & h_{m_l}\eta_l^{l-1} \end{pmatrix} \begin{pmatrix} e_{m_1} \\ e_{m_2} \\ e_{m_3} \\ \vdots \\ e_{m_l} \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{l-1} \end{pmatrix}$$

and then return $y - e$ (or *Error* if one or more of the components of e is not in K).

3.6 Remark. Even with the improvements advanced in this note, in theory the PGZ and PGZm algorithms cannot beat, for very large alternant codes, the Berlekamp-Massey-Sugiyama (BMS) algorithm (cf. [4], Section 4.3). But they are comparable for the codes that are feasible in practice. Indeed, the very construction of the alternant matrix is costly in time and space and within the range of parameters that can usually be afforded, the efficiency of the PGZ or PGZm is comparable to that of BMS. Let us also say that in some contexts, as for example in teaching, the PGZm has the advantage that it is more straightforward to explain and to implement, the easiest case being RS codes over $K = \mathbb{F}_p$, p prime.

4. Examples

Here we are going to discuss the implementation of the algorithms using [3], and how it works, by considering some examples for each of the following classes: RS, GRS, BCH and (classical) Goppa codes.

In the code constructors described below, **h** and **a** stand for variables bound to vectors of the same length n with entries in a finite field; **K** and **F**, to finite fields K and F ; **r**, **k**, **d** and **l**, to integers r , k , d and l used as in the first two sections; and **g** to a univariate polynomial with coefficients in a finite field.

- **AC(h,a,r,K)**: This constructs the alternating code $A_K(\mathbf{h}, \boldsymbol{\alpha}, r)$. In the context of this note, it is our main constructor, as the others (described below) are in fact defined as special calls to **AC** (cf. Section 1).
- **RS(a,k)**: This yields the RS code $\text{RS}(\boldsymbol{\alpha}, k)$, an $[n, k, n - k + 1]$ code defined over the field to which the elements of $\boldsymbol{\alpha}$ belong.
- **GRS(h,a,k)**: As **RS**, but we have to supply **h** as a first argument.

- **PRS(F,k)**: The primitive RS code of the finite field F . It is defined as **RS(a,k)**, but taking as α the list of non-zero elements of F .
- **BCH(a,d,l)**: Supplies the code **BCH**(α, d, l), where here \mathbf{a} stands for an element α in a finite field.
- **Goppa(g,a)**: The Goppa code $\Gamma(g, \alpha)$.

The code C obtained by any one of these constructors is a record-like structure with fields that allow to get data from the code or store new information about it. The labels of those fields end with an underscore, but otherwise tend to mimic the mathematical symbols. For example, $\mathbf{a} = \mathbf{a_}(C)$ and $\mathbf{H} = \mathbf{H_}(C)$ bind the variable \mathbf{a} to the vector α and the variable \mathbf{H} to the alternant control matrix of C .

4.1 Remark. Except for RS and GRS codes, for which the parameters can be deduced immediately from the data supplied to their constructors, in general there is some work to be done to determine those not yet known. This work is rather straightforward when it comes to compute $k = \dim_K C$. To that end, we need to construct a control matrix of C defined over K . This can be done in two steps: replace each entry of H by the column of its components in the natural linear basis of \bar{K} over K (this yields an K -control matrix, but it usually has redundant rows) and then suppress all the rows that are linear combinations of the previous ones. We have implemented these steps by means of the functions **blow**(H, K) and **prune**(M). The bottom line is that the dimension of C is $n - r'$, where r' is the number of rows of **prune**(**blow**(H, K)) or just **rank**(**blow**(H, K)). This is the method used to determine the dimension k and the rate $R = k/n$ when we quote them.

A final comment before getting into the examples is that we can assume that the received vector is an error vector e such that $\text{wt}(e) \leq t$. The reason is the linearity of the code, which implies that only the error vector is involved in the computations of the error positions and values.

RS. Take $K = \mathbb{F}_{13}$ and construct $C = \text{PRS}(K, 8)$, the primitive RS of K of dimension $k = 8$. It has length $n = 12$, so its rate is $2/3$, and the minimum distance is $d = n - k + 1 = 5$, so that it corrects at least two errors. First let us consider the case of one error. Suppose the received vector is

$$e = [0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0]$$

Then the decoder call **PGZ**(e, C) (or **PGZm**(e, C)) yields

```
PGZ: Error positions [4], error values [3] :: Vector[Z13]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] :: Vector[Z13]
```

This means that PGZ finds that there is a single error at the index 4 (5th element of the vector) and that its value is 3, and then outputs (correctly) the decoded vector. Let us go over the steps followed by PGZ in detail. The control matrix is $H = H_-(C)$:

```
[[1  2  4  8  3  6 12 11  9  5 10  7]
 [1  4  3 12  9 10  1  4  3 12  9 10]
 [1  8 12  5  1  8 12  5  1  8 12  5]
 [1  3  9  1  3  9  1  3  9  1  3  9]] :: Matrix[Z13]
```

Then the syndromy vector is given by $s = y * \text{transpose}(H)$,

```
[9, 1, 3, 9],
```

and the matrix $S = \text{hankel_matrix}(s)$ is

```
[[9 1 3]
 [1 3 9]] :: Matrix[Z13]
```

This matrix has rank 1, as the second row is 3 times the first. This means that there is one error and that the error-locating polynomial is $L(z) = z - 1/9 = z - 3$. To find the error position, we have to look at the position of 3 in the vector α used to construct C , which is $a_-(C)$:

```
[1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7] :: Vector[Z13]
```

Thus the position is indeed the one in which the error occurred. To find the error value, first we have to calculate the error evaluator

$$E(z) = \sigma(z) \tilde{L}(z) \mod z^r = (9 + z + 3z^2 + 9z^3)(-3z + 1) \mod z^6,$$

which turns out to be the constant 9. Forney's formula for the error value is $-\alpha_4 E(1/\alpha_4) / h_4 \tilde{L}'(1/\alpha_4) = -3 \cdot 9/3 \cdot (-3) = 3$ (for in this case $\mathbf{h} = \alpha$), which is the error value.

Now we are going to repeat, with less detail, the case of 2 errors. Suppose the received vector is

```
y = [0, 0, 0, 0, 3, 0, 0, 0, 0, 7, 0, 0]
```

Then the decoder call $\text{PGZ}(y, C)$ (or $\text{PGZm}(y, C)$) yields

```
PGZ: Error positions [4, 9], error values [3, 7] :: Vector[Z13]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] :: Vector[Z13]
```

The syndromy vector is

```
[5, 7, 7, 3],
```

and the matrix $S = \text{hankel_matrix}(s)$ is

```
[[5 7 7]
 [7 7 3]] :: Matrix[Z13]
```

Since it has rank 2, there have been 2 errors. In this case the Gauss-Jordan reduction produces $L(z) = z^2 + 5z + 2 = (z - 3)(z - 5)$. Since the roots 3 and 5 occupy the positions 5 and 10 in α , we see that the indices of the computed error positions are correct. For the error values we have to apply Forney's formula to 3 and 5 (α_4 and α_9). We have $\tilde{L}(z) = 2z^2 + 5z + 1$, $\tilde{L}'(z) = 4z + 5$, $\sigma(z) = 5 + 7z + 7z^2 + 3z^3$, and $E(z) = \tilde{L}(z)\sigma(z) \bmod z^4 = 6z + 5$. Then the error corresponding to, for example, the second root is

$$-5 \cdot E(1/5)/5 \cdot \tilde{L}'(1/5) = -E(8)/\tilde{L}'(8) = -(48 + 5)/(32 + 5) = -1/-2 = 7.$$

For another example, if we take $F = \text{Zn}(31)$ and $k = 20$, then $C = \text{PRS}(F, k)$ is a $[30, 20, 11]$ code. This corrects up to 5 errors and its rate is $2/3$. This capability is illustrated in the following listing:

```
e = rd_error_vector(F,n,5) # this creates a random 5-error pattern
>>[0,0,0,0,0,0,0,0,0,14,0,0,0,28,26,0,0,0,23,0,0,16,0,0,0,0,0,0]
:: Vector[Z31]
PGZ(e,C)
>>PGZ: Error positions [9,13,14,19,22],
      error values [14,28,26,23,16] :: Vector[Z31]
```

BCH. Take $K = \mathbb{F}_2$ and $F = \mathbb{F}_{32}$, generated by α such that $\alpha^5 = \alpha^2 + 1$. Let $C = \text{BCH}(\alpha, 7)$. This is a binary code of length $n = 31$ (the order of α) that corrects up to 3 errors. In our system it can be constructed as follows:

```
K = Zn(2)
[F,a] = extension(K,[1,0,0,1,0,1], 'a', 'F')
C = BCH(a,7)
```

Its dimension is 16 (so its rate is $16/31 > 1/2$), as shown by the following command:

```
n - rank(Blow(H_(C),K))
>> 16
```

Now consider, for example, the weight 3 error pattern e :

```
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
```

Then the call `PGZ(e,C)` outputs

```
PGZ: Error positions [5, 19, 28], error values [1, 1, 1] :: Vector[K]
```

(Note that the only possible error value over \mathbb{F}_2 is 1, and that therefore in this case the decoder only needs to care about error location.) The matrix S computed in this case is (instead of α^j we write j):

```
[[22, 13, 14, 26]
 [13, 14, 26, 19]
 [14, 26, 19, 28]]
```

which gives $l = 3$.

The code C also corrects up to 3 errors when considered as an F -code (which is a GRS code over F). For example, if

```
e=[0,0,0,0,0,0,0,0,  $\alpha^5$ , 1, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  $\alpha^{19}$ , 0,0,0,0]
```

then the output contains

```
PGZ: Error positions [8, 9, 26],
      error values [a**5, 1, a**19] :: Vector[F]
```

together with the correct decoded vector. With the same conventions as before, the matrix S gives $l = 3$:

```
[[16, 0, 30, 14]
 [ 0, 30, 14, 25]
 [30, 14, 25, 28]]
```

Here is a more involved example. Take $K = \mathbb{F}_3$ and $F = \mathbb{F}_{343}$, generated by α such that $\alpha^5 = \alpha + 2$. The element α is primitive and $\beta = \alpha^2$ has order $n = (343 - 1)/2 = 121$. It follows that the code $C = \text{BCH}(\beta, 11)$ has length n and that it corrects at least 5 errors. In the PyCC system it can be constructed as follows:

```
K = Zn(3)
f=get_irreducible_polynomial(K,5,'X') # X**5-X+1
[F,a] = extension(K,f,'a','F')
C = BCH(a**2,11)
```

In addition, the command `n - rank(blow(H_(C),K))` yields that its dimension is $121 - 35 = 86$, so that its rate is $86/121 > 7/10$.

Consider the error pattern of weight 5 (where 0_k denotes 0 repeated k times)

```
e = [02, 1, 07, 1, 022, 2, 06, 2, 072, 1, 07] :: Vector[Z3]
```

Then we have:

```
PGZ(e,C)
>>PGZ: Error positions [2, 10, 33, 40, 113],
      error values [1, 1, 2, 2, 1] :: Vector[F5]
```

Classical Goppa. Consider the field \mathbb{F}_{25} generated over \mathbb{F}_5 by x such that $x^2 = 2$. Let $g = T^6 + T^3 + T + 1$ and make a list α of the elements $t \in \mathbb{F}_{25}$ such that $g(t) \neq 0$. Then it turns out that α has length $n = 19$ (g has four simple roots and one double root in \mathbb{F}_{25}) and that $C = \Gamma(g, \alpha)$ corrects up to 3 errors.

```
F5 = Zn(5)
# Creation of F25, with generator x
[F25,x] = extension(F5,[1,0,-2], 'x', 'F25')
# Creation of the polynomial ring F25[T]
[A,T] = polynomial_ring(F25,'T')

g = T**6 + T**3 + T + 1
a = Set(F25)[1:] # The non-zero elements of F25
a = [t for t in a if evaluate(g,t)!=0]
C = Goppa(g,a)

# generate a random error pattern of weight 3
e = rd_error_vector(Z5,n,3)
>> e = [0,1,0,0,0,3,0,4,0,0,0,0,0,0,0,0,0,0,0] :: Vector[Z5]

# Use the PGZ decoder for C
PGZ(e,C)
>>PGZ: Error positions [1,5,7], error values [1,3,4] :: Vector[K]
      [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] :: Vector[Z5]
```

The dimension of C (given by $n - \text{rank}(\text{blow}(H_-(C), F5))$) is $19 - 12 = 7$, and its rate $7/19 > 1/3$.

Here is another illustration, with $n = 76$:

```
e = rd_error_vector(F3,n,5) >> [010, 2, 035, 2, 09, 1, 06, 1, 03, 2, 010]

F3 = Zn(3)
f = get_irreducible_polynomial(F3,4,'X') # X**4 + X + 2
[F81,x] = extension(F3,f,'x', 'F81')      # x is primitive
g = X**2 * (X-1)**4 * (X-2)**4
a = Set(F81)[3:] # g does not vanish on any of these values
n = len(a)      # 81 - 3 = 76
C = Goppa(g,a)  # code of length 76
k = n - rank(blow(H_-(C),F3)) # k = 76-32 = 44
PGZm(e,C)
>> PGZm: Error positions [10, 46, 56, 63, 67],
      error values [2, 2, 1, 1, 2] :: Vector[F3]
```

Appendix A. The function PGZm

For the sake of brevity, here we list and comment the $\text{PGZm}(y, C)$ function. Its code, together with the code of $\text{PGZ}(y, C)$, can be accessed by browsing the text file Listings-FSX.pycc. The parameter y is supposed to be the received vector in a transmission using the alternant code C . We have seen that the value of expressions $a_ (C)$ and $H_ (C)$ is the vector α and the control matrix H . Similarly, the values of the expressions $K_ (C)$, $h_ (C)$, $r_ (C)$ are the field over which C is defined, the vector h and the number of rows of H , respectively.

```
def PGZm(y,C):
    if isinstance(y,list): y = vector(K_(C),y)
    if not isinstance(y,Vector_Element):
        return "PGZm: Argument is not a vector"
    h = h_(C)
    if len(y) != len(h):
        return "PGZm: Vector argument has wrong length"
    r = r_(C); alpha = a_(C); H = H_(C); K = K_(C)
    s = y*H.transpose()
    if is_zero(s):
        print("PGZm: Input is a code vector")
        return y
    S = hankel_matrix(s)
    c0 = S[:,0] # keep the first column of S
    a = -GJ(S); l = len(a)
    a = reverse(a.to_list())
    K1 = K_(H)
    [_,z] = polynomial_ring(K1,'z','K1[z]')
    L = hohner([1]+a,z)
    R = [s for s in alpha if evaluate(L,s)==0]
    if len(R) < 1:
        return "PGZm: Defective error location"
    M = [alpha.to_list().index(r) for r in R]
    h1 = [h[m] for m in M]
    V = alternant_matrix(h1,R,l)
    v = c0[:l]
    V1 = splice(V,v)
    w = transpose(GJ(V1))
    for t in w:
        t = pull(t,K)
        if not belongs(t,K):
            return "PGZ: error value not in base field"
    show("PGZm: Error positions {}, error values {}".format(M, w))
```

```

for j in range(len(M)):
    y[M[j]]-=w[j]
return pull(y,K)

```

Appendix B. The PyCC system

Initially (October 2015) the idea that launched [3] was to match the functionality of the CC system developed to deal with the computational tasks related to the book [4], but it became soon clear that we could go beyond that system in several directions. The aim of the undertaking is to produce a Python package (PyCC) enabling the construction, coding and decoding of error-correcting codes and make it freely available for teachers and researchers. The current state of the project is documented at PyCC

References

- [1] R. Farré. Notes on information theory and coding theory, 2003. In Catalan.
- [2] W. W. Peterson and E. J. Weldon. *Error-Correcting codes*. MIT Press (2nd edition), 1972.
- [3] N. Sayols and S. Xambó-Descamp. A Python package for the construction, coding and decoding of error-correcting codes. PyCC, 2017.
- [4] S. Xambó-Descamps. *Block error-correcting codes: a computational primer*. Univesitext. Springer, 2003.